Introduction to the SAM User Language

National Renewable Energy Laboratory

March 14, 2011

Abstract

The SAM User Language (SamUL) is a built-in scripting language that allows a user to automate tasks and perform more complex analyses directly from within SAM. This guide assumes some rudimentary facility with basic programming concepts and familiarity with the SAM interface, capabilities, and general work flow.

The System Advisor Model (SAM) provides a consistent framework for analyzing and comparing system performance and costs across a wide range of renewable energy technologies and markets, from residential photovoltaic systems to utility scale concentrating solar power plants.

System Advisor is based on an hourly simulation engine that interacts with performance, cost, and finance models to calculate energy output, costs, and cash flows, including the effect of incentives.

Contents

1 Introduction

1.1 Why use SamUL?

Suppose you are an energy analyst, and your client asks for a custom map of the United States showing the levelized cost of energy (LCOE) at several hundred locations in the country for a specific photovoltaic system. Since you are familiar with the Solar Advisor Model, and have access to weather data for all the requested locations, the task falls to you to generate the LCOE values for the custom map. With the PV system specifications, you succeed in setting up a "base case" simulation in SAM for one location.

You could run a SAM analysis for each weather file individually, recording the results in a spreadsheet. You could even set up many locations in a parametric simulation. Either way would be tedious and error prone at best for several hundred iterations. Even worse, when your client decides to change even just one specification of the system, you would have to start all over.

As others have done in the past, you could extract the generated PV simulator setup files, and write a program in C or Excel/VBAScript to call the TRNSYS simulation engine that sits behind SAM to loop over the various weather files. This too proves troublesome, because if any specifications change, you would have to edit the complicated TRNSYS input files by hand. Furthermore, processing the large amounts of data in the system performance output files would at best be an unpleasant chore. The worst part is that this "solution" does not even give you the LCOE at the end, because that is calculated inside the SAM financial model not TRNSYS!

The SAM User Language was designed to solve these problems. It lets you perform this otherwise cumbersome LCOE task in no more than 10 lines of code. When a system specification changes, or new weather data is available, you only need to re-run the script after modifying the appropriate inputs in SAM. Even better, there is no need for any additional software or expertise. This guide will help you learn the basics of SamUL programming in the hope that it will save you significant time and effort in the future.

1.2 Entering a SamUL Script

SamUL is included by default in every installation of the Solar Advisor Model, and is accessed from the 'Developer' menu in SAM.

Since SamUL files are saved as part of a SAM project, you must start a new project or open one of the included sample files to begin. Once your project is open, you can create a new script from "Developer:New SamUL Script". A tab will open showing an empty text editor and a toolbar with a few buttons.

The first button will interpret and run the SamUL script that is in the text editor. If there are any syntax errors, SAM will display an error message box and force you to correct any errors before it runs the script. If you have typed something incorrectly, the first error location displayed is nearest the point in the source code where the typo occurred.

If your code is syntactically correct, SAM will run it and display a console dialog to capture any output that your script may create.

1.3 Hello world!

As with any new language, the traditional first program is to print the words "Hello, world!" on the screen, and the SamUL version is listed below.

```
out( "Hello, world!\n" )
```

Notable features:

1. The `out` command generates text output from the script
2. The text to be printed is enclosed in double-quotes
3. To move to a new line in the output window, the symbol `\n` is used

To run Hello world, type it into a new SamUL script, and press the 'Run' button on the toolbar.

1.4 Why SamUL instead of VBA?

Since SamUL is so similar in functionality and syntax to Visual Basic, you might wonder why we didn't simply put a VBA engine behind SAM. Visual Basic for Applications (VBA) is a closed source Microsoft product that is very tightly integrated into the Office suite of applications, and while there are some freely available interpreters for subsets of VB-like languages, we decided that engineering a simple scripting language would be straightforward enough. In the future, we might consider integrating other well known languages, such as Lua.

In general, SamUL is close enough to VBA in syntax and structure to make it easily understandable by people familiar with VBA. By developing our own script engine, we have been able to integrate it very tightly into the SAM environment for maximum ease of use.

Some notable points distinctions include:

1. '`for`' loop syntax follows the C / Perl convention
2. Array arithmetic is automatically performed element by element
3. '`elseif`' statement formatting is similar to PHP
4. No distinction between functions and procedures

2 Data Variables

2.1 General Syntax

In SamUL, a program statement is generally placed on a line by itself, and the end-of-line marks the end of the statement. Currently, there is no facility to split a long statement across multiple lines.

Blank lines may be inserted between statements. While they have no meaning, they can help make a script easier to read. Spaces can also be added or removed nearly anywhere, except of course in the middle of a word. The following statements all have the same meaning.

```
out("Hello 1\n")
  out  ("Hello 2\n")
out (  "Hello 3\n"  )
```

Comments are lines in the program code that are ignored by SamUL. They serve as a form of documentation, and can help other people (and you!) more easily understand what the script does. Comments begin with the single-quote `'` character, and continue to the end of the line.

```
' this program creates a greeting
out( "Hello, world!\n" ) ' display the greeting to the user
```

2.2 Variables

Variables store information while your script is running. SamUL variables share many characteristics with other computer languages.

1. Variables do not need to be "declared" in advance of being used
2. There is no distinction between variables that store text and variables that store numbers

Variable names may contain letters, digit, and the underscore symbol. A limitation is that variables cannot start with a digit. Unlike some languages like C and Perl, SamUL does not distinguish between upper and lower case letters in a variable (or subroutine) name. As a result, the name `myData` is the same as `MYdata`.

Values are assigned to variables using the equal sign `=`. Some examples are below.

```
Num_Modules = 10
ArrayPowerWatts = 4k
Tilt = 18.2
system_name = "Super PV System"
Cost = "unknown"
COST = 1e6
cost = 1M
```

Assigning to a variable overwrites its previous value. As shown above, decimal numbers can be written using scientific notation or engineering suffixes. The last two assignments to `Cost` are the same value. Recognized suffixes are listed in the table below. Suffixes are case-sensitive, so that SamUL can distinguish between `m` (milli) and `M` (Mega).

2.3 Arithmetic

SamUL supports the four basic operations `+`, `-`, `*`, and `/`. The usual algebraic precendence rules are followed, so that multiplications and divisions are performed before additions and subtractions. Parentheses are also understood and can be used to change the default order of operations.

Name	Suffix	Multiplier
Tera	T	1e12
Giga	G	1e9
Mega	M	1e6
Kilo	k	1e3
Milli	m	1e-3
Micro	u	1e-6
Nano	n	1e-9
Pico	p	1e-12
Femto	f	1e-15
Atto	a	1e-18

Table 1: Recognized Numerical Suffixes

Operators are left-associative, meaning that the expression `3-10-8` is understood as `(3-10)-8` .

More complicated operations like raising to a power and performing modulus arithmetic are possible using built-in function calls in the standard SamUL library.

Examples of arithmetic operations:

```
battery_cost = cost_per_kwh * battery_capacity

' multiplication takes precedence
degraded_output = degraded_output - degraded_output * 0.1

' use parentheses to subtract before multiplication
cash_amount = total_cost * ( 1 - debt_fraction/100.0 )
```

2.4 Simple Input and Output

You can use the built-in `out` and `outln` functions to write data to the console window. The difference is that `outln` automatically appends a newline character to the output. To output multiple text strings or variables, use the `+` operator, or separate them with a comma.

```
array_power = 4.3k
array_eff = 0.11
outln("Array power is " + array_power + " Watts.")
outln("It is " + (array_eff*100) + " percent efficient.")
outln("It is ", array_eff*100, " percent efficient.") ' same as above
```

The console output generated is:

```
Array power is 4300 Watts.
It is 11 percent efficient.
```

Use the `in` function to read input from the user. You can optionally pass a message to `in` to display to the user when the input popup appears. The user can enter either numbers or text, and SamUL will perform any type conversions if needed (and if possible).

```
cost_per_watt = in("Enter cost per watt:")  ' Show a message.  in() also is fine.
notice( "Total cost is: " + cost_per_watt * 4k + " dollars") ' 4kW system
```

The `notice` function works like `out`, except that it displays a pop up message box on the computer screen.

2.5 Data Types and Conversion

SamUL supports four basic types of data, although most conversions between types happen automatically. Because of this, SamUL is generally a weakly typed language, meaning that you can add text variables to number variables, and SamUL with try to make a appropriate conversion in context.

Type	Conversion Function	Valid Values
Integer Number	`integer()`	+/- approx. 2 billion
Double-precision Decimal Number	`double()`	1e-308 to 1e308, with infinity
Boolean	`boolean()`	`true` or `false` (`1` or `0`)
Text Strings	`string()`	Any length text string

Table 2: Intrinsic SamUL Data Types

Sometimes you have two numbers in text strings that you would like to multiply. This can happen if you read data in from a text file on the computer, for example. Since it does not make sense to try to multiply text strings, you need to first convert the strings to numbers. To convert a variable to a double-precision decimal number, use the `double` function, as below.

```
a = "3.5"
b = "-2"
c1 = a*b ' this will cause an error when you click 'Run'
c2 = Double(a) * Double(b) ' this will assign c2 the number value of -7
```

The assignment to `c1` above will cause the error *Error: Invalid string operator '*'*, while the assignment to `c2` makes sense and executes correctly.

You can also use `integer` to convert a string to an integer or truncate a decimal number, or the `string` function to explicitly convert a number to a string variable.

If you need to find out what type a variable currently has, use the `typeof` function to get a description.

```
a = 3.5
b = -2
c1 = a+b ' this will set c1 to -1.5
c2 = String( Integer(a) ) + String( b ) ' c2 set to text "3-2"
```

```
outln( typeof(a) ) ' will display "double"
outln( typeof(c2) ) ' will display "string"
```

2.6 Special Characters

Text data can contain special characters to denote tabs, line endings, and other useful elements that are not part of the normal alphabet. These are inserted into quoted text strings with *escape sequences*, which begin with the \ character.

Escape Sequence	Meaning
`\n`	New line
`\t`	Tab character
`\r`	Carriage return
`\"`	Double quote
`\\`	Backslash character

Table 3: Text String Escape Sequences

So, to print the text `"Hi, tabbed world!"`, or assign `c:\Windows\notepad.exe`, you would have to write:

```
outln("\"Hi,\ttabbed world!\"")
program = "c:\\Windows\\notepad.exe"
```

Note that for file names on a Windows computer, it is important to convert back slashes (`'\'`) to forward slashes (`/`). Otherwise, the file name may be translated incorrectly and the file won't be found.

3 Flow Control

3.1 Comparison Operators

SamUL supports many ways of comparing data. These types of tests can control the program flow with branching and looping constructs that we will discuss later.

There are six standard comparison operators that can be used on most types of data. For text strings, "less than" and "greater than" are with respect to alphabetical order.

Examples of comparisons:

```
divisor != 0
state == "oregon"
error <= -0.003
"pv" > "csp"
```

Single comparisons can be combined by *boolean* operators into more complicated tests.

Comparison	Operator
Equal	`==`
Not Equal	`!=`
Less Than	`<`
Less Than or Equal	`<=`
Greater Than	`>`
Greater Than or Equal	`>=`

Table 4: Comparison Operators

1. The `not` operator yields true when the test is false. It is placed before the test whose result is to be notted.
   Example: `not (divisor == 0)`
2. The `and` operator yields true only if both tests are true.
   Example: `divisor != 0 and dividend > 1`
3. The `or` operator yields true if either test is true.
   Example: `state == "oregon" or state == "colorado"`

The boolean operators can be combined to make even more complex tests. The operators are listed above in order of highest precedence to lowest. If you are unsure of which test will be evaluated first, use parentheses to group tests. Note that the following statements have very different meanings.

```
state_count > 0 and state_abbrev == "CA" or state_abbrev == "OR"
state_count > 0 and (state_abbrev == "CA" or state_abbrev == "OR")
```

3.2 Branching

Using the comparison and boolean operators to define tests, you can control whether a section of code in your script will be executed or not. Therefore, the script can make decisions depending on different circumstances and user inputs.

3.2.1 `if` Statements

The simplest branching construct is the `if` statement. For example:

```
if ( tilt < 0.0 )
    outln("Error: tilt angle must be 0 or greater")
end
```

Note the following characteristics of the `if` statement:

1. The test is placed in parentheses after the `if` keyword.
2. The following program lines include the statements to execute when the `if` test succeeds.
3. To help program readability, the statements inside the `if` are usually indented.

4. The construct concludes with the `end` keyword.
5. When the `if` test fails, the program statements inside the `if-end` block are skipped.

3.2.2 `else` Construct

When you also have commands you wish to execute when the `if` test fails, use the `else` clause. For example:

```
if ( power > 0 )
    energy = power * time
    operating_cost = energy * energy_cost
else
    outln("Error, no power was generated.")
    energy = -1
    operating_cost = -1
end
```

3.2.3 Multiple `if` Tests

Sometimes you wish to test many conditions in a sequence, and take appropriate action depending on which test is successful. In this situation, use the `elseif` clause. Be careful to spell it as a single word, as both `else if` and `elseif` can be syntactically correct, but have different meanings.

```
if ( angle >= 0 and angle < 90)
    text = "first quadrant"
elseif ( angle >= 90 and angle < 180 )
    text = "second quadrant"
elseif ( angle >= 180 and angle < 270 )
    text = "third quadrant"
else
    text = "fourth quadrant"
end
```

You do not need to end a sequence of `elseif` statements with the `else` clause, although in most cases it is appropriate so that every situation can be handled. You can also nest `if` constructs if needed. Again, we recommend indenting each "level" of nesting to improve your script's readability. For example:

```
if ( angle >= 0 and angle < 90 )
    if ( print_value == true )
        outln( "first quadrant: " + angle )
    else
        outln( "first quadrant" )
    end
end
```

3.2.4 Single line `if`s

Sometimes you only want to take a single action when an `if` statement succeeds. To reduce the amount of code you must type, SamUL accepts single line `if` statements, as shown below.

```
if ( azimuth < 0 ) outln( "Warning: azimuth < 0, continuing..." )

if ( tilt > 90 ) tilt = 90   ' set maximum tilt value
```

You can also use an `else` statement on single line `if`. Like the `if`, it only accepts one program statement, and must be typed on the same program line. Example:

```
if ( value > average ) outln("Above average") else outln("Not above average")
```

3.3 Looping

A loop is a way of repeating the same commands over and over. You may need to process each line of a file in the same way, or sort a list of names. To achieve such tasks, SamUL provides two types of loop constructs, the `while` and `for` loops.

Like `if` statements, loops contain a "body" of program statements followed by the `end` keyword to denote where the loop construct ends.

3.3.1 `while` Loops

The `while` loop is the simplest loop. It repeats one or more program statements as long as a logical test holds true. When the test fails, the loop ends, and the program continues execution of the statements following the loop construct. For example:

```
while ( done == false )
    ' process some data
    ' check if we are finished and update the 'done' variable
end
```

The test in a `while` loop is checked before the body of the loop is entered for the first time. In the example above, we must set the variable `done` to `false` before the loop, because otherwise no data processing would occur. After each iteration ends, the test is checked again to determine whether to continue the loop or not.

3.3.2 Counter-driven Loops

Counter-driven loops are useful when you want to run a sequence of commands for a certain number of times. As an example, you may wish to display only the first 10 lines in a text file.

There are four basic parts of implementing a counter-driven loop:

1. Initialize a counter variable before the loop begins.

2. Test to see if the counter variable has reached a set maximum value.
3. Execute the program statements in the loop, if the counter has not reached the maximum value.
4. Increment the counter by some value.

For example, we can implement a counter-driven loop using the `while` construct:

```
i = 0       ' use i as counter variable
while (i < 10)
    outln( "value of i is " + i )
    i = i + 1
end
```

3.3.3 `for` Loops

The `for` loop provides a streamlined way to write a counter-driven loop. It combines the counter initialization, test, and increment statements into a single line. The script below produces exactly the same effect as the `while` loop example above.

```
for ( i = 0; i < 10; i = i+1 )
    outln( "value of i is " + i )
end
```

The three loop control statements are separated by semicolons in the `for` loop statement. The initialization statement (first) is run only once before the loop starts. The test statement (second) is run before entering an iteration of the loop body. Finally, the increment statement is run after each completed iteration, and before the test is rechecked. Note that you can use any assignment or calculation in the increment statement.

Just like the `if` statement, SamUL allows `for` loops that contain only one program statement in the body to be written on one line. For example:

```
for ( val=57; val > 1; val = val / 2 ) outln("Value is " + val )
```

3.3.4 Loop Control Statements

In some cases you may want to end a loop prematurely. Suppose under normal conditions, you would iterate 10 times, but because of some rare circumstance, you must break the loop's normal path of execution after the third iteration. To do this, use the `break` statement.

```
value = double( in("Enter a starting value") )
for ( i=0; i<10; i=i+1 )
    outln("Value is " + value )
    if (value < 0)
        break
    end
    value = value / 3.0
```

```
end
```

In another situation, you may not want to altogether break the loop, but skip the rest of program statements left in the current iteration. For example, you may be processing a list of files, but each one is only processed if it starts with a specific line. The `continue` keyword provides this functionality.

```
for ( i=0; i<file_count; i=i+1 )
    file_header_ok = false

    ' check if whether current file has the correct header

    if (file_header_ok == false)
        continue
    end

    ' process this file
end
```

The `break` and `continue` statements can be used with both `for` and `while` loops. If you have nested loops, the statements will act in relation to the nearest loop structure. In other words, a `break` statement in the body of the inner-most loop will only break the execution of the inner-most loop.

3.4 Quitting

SamUL script execution normally ends when there are no more statements to run at the end of the script. However, sometimes you may need to halt early, if the user chooses not to continue an operation.

The `exit` statement will end the SamUL script immediately. For example:

```
if ( yesno("Do you want to quit?") == true )
    outln("Aborted.")
    exit
end
```

The `yesno` function call displays a message box on the user's screen with yes and no buttons, showing the given message. It returns `true` if the user clicked yes, or `false` otherwise.

4 Arrays of Data

Often you need to store a list of related values. For example, you may need to refer to the price of energy in different years. Or you might have a table of state names and capital cities. In SamUL, you can use arrays to store these types of collections of data.

4.1 Initializing and Indexing

An *array* is simply a list of variables that are indexed by numbers. Each variable in the array is called an *element* of the array, and the position of the element within the array is called the element's *index*. The index of the first element in an array is always 0.

To access array elements, enclose the index number in square brackets immediately following the variable name. SamUL does not require you to declare or allocate space for the array data in advance.

```
names[0] = "Sean"
names[1] = "Walter"
names[2] = "Pam"
names[3] = "Claire"
names[4] = "Patrick"

outln( names[3] ) ' output is "Patrick"
my_index = 2
outln( names[my_index] ) ' output is "Pam"
```

You can also initialize a fixed array using the `array` command provided in SamUL. Simply separate each element with a comma. There is no limit to the number of elements you can pass to `array`.

```
names = array("Sean", "Walter", "Pam", "Claire", "Patrick")
outln( "First: " + names[0] )
outln( "All: " + names )
```

Note that calling the `typeof` function on an array variable will return "array" as the type description, not the type of the elements. This is because SamUL is not strict about the types of variables stored in an array, and does not require all elements to be of the same type.

4.2 Array Length

Sometimes you do not know in advance how many elements are in an array. This can happen if you are reading a list of numbers from a text file, storing each as an element in an array. After the all the data has been read, you can use the `length` function to determine how many elements the array contains.

```
count = length( names )
```

4.3 Processing Arrays

Arrays and loops naturally go together, since frequently you may want to perform the same operation on each element of an array. For example, you may want to find the total sum of an array of numbers.

```
numbers = array( 1, -3, 2.4, 9, 7, 22, -2.1, 5.8 )

count = length( numbers )
sum = 0
for (i=0; i<count; i=i+1)
    sum = sum + numbers[i]
end
```

The important feature of this code is that it will work regardless of how many elements are in the array.

4.4 Multidimensional Arrays

As previously noted, SamUL is not strict with the types of elements stored in an array. Therefore, a single array element can even be another array. This allows you to define matrices with both row and column indexes, and also three (or greater) dimensional arrays.

To create a multi-dimensional array, simply separate the indices with commas between the square brackets. For example:

```
data[0,0] = 3
data[0,1] = -2
data[1,0] = 5
data[2,0] = 1

nrows = length(data) ' result is 4
ncols = length(data[0]) ' result is 2

row1 = data[0]  ' extract the first row

x = row1[0] ' value is 3
y = row1[1] ' value is -2
```

4.5 Managing Array Storage

When you define an array, SamUL automatically allocates sufficient computer memory to store the elements. If you know in advance that your array will contain 100 elements, for example, it can be much faster to allocate the computer memory before filling the array with data. Use the `allocate` command to make space for 1 or 2 dimensional arrays.

```
data = allocate(3,2)  ' a matrix with 3 rows and 2 columns
data[2,1] = 3

prices = allocate( 5 ) ' a simple 5 element array
```

As before, you can extend the array simply by using higher indexes. However, if you know in advance how many more elements you will be adding, it can be faster to use the `resize` command

to reallocate computer memory to store the array. `resize` preserves any data in the array, or truncates data if the new size is smaller than the old size.

```
data = allocate(5)
outln( length(data) )
resize(data, 10)
outln( length(data) )

resize(data, 2, 4)
outln( length(data) )
outln( length( data[0] ) )
```

4.6 Multiple Advance Declarations

You can also declare many variables and arrays in advance using the `declare` statement. For example:

```
declare radiation[8760],temp[8760],matrix[3,3],i=0
```

This statement will create the array variables `radiation` and `temp`, each with 8760 values. It will also set aside memory for the 3x3 `matrix` variable, and 'create' the variable `i` and assign it the value of zero. The `declare` statement can be a useful shortcut to creating arrays and initializing many variables in a single line. The only limitation is that you cannot define arrays of greater than two dimensions using the `declare` command.

5 Functions Calls

It is usually good programming practice to split a larger program up into smaller sections, often called procedures, functions, or subroutines. A program may be easier to read and debug if it is not all thrown together, and you may have common blocks of code that appear several times in the program.

5.1 User Functions

A function is simply a named chunk of code that may be called from other parts of the script. It usually performs a well-defined operation on a set of variables, and it may return a computed value to the caller.

Functions can be written anywhere in your SAM script, including after they are called. If a function is never called by the program, it has no effect.

5.1.1 Definition

Consider the very simple procedure listed below.

```
function show_welcome()
   outln("Thank you for choosing SamUL.")
   outln("This text will only be displayed at the start of the script.")
end
```

Notable features:

1. Use the `function` keyword to define a new function.
2. The function name is next, and follows the same rules as for variable names. Valid function names can have letters, digits, and underscores, but cannot start with a digit.
3. The empty parentheses after the name indicate that this function takes no parameters.
4. The `end` keyword closes the function definition.

To call the function from elsewhere in the code, simply write the function's name, followed by the parentheses.

```
' show a message to the user
show_welcome()
```

5.1.2 Returning a Value

A function is generally more useful if it can return information back to the program that called it. In this example, the function will not return unless the user enters "yes" or "no" into the input dialog.

```
function require_yes_or_no()
   while( true )
      answer = in("Destroy everything? Enter yes or no:")
      if (answer == "yes") return true
      if (answer == "no") return false
      outln("That was not an acceptable response.")
   end
end

' call the input function
result = require_yes_or_no() ' returns true or false
if ( not result )
   outln("user said no, phew!")
   exit
else
   outln("destroying everything...")
end
```

The important lesson here is that the main script does not worry about the details of how the user is questioned, and only knows that it will receive a `true` or `false` response. Also, the function can be reused in different parts of the program, and each time the user will be treated in a familiar way.

5.1.3 Parameters

In most cases, a function will accept arguments when it is called. That way, the function can change its behavior, or take different inputs in calculating a result. Analogous to mathematical functions, SamUL functions can take arguments to compute a result that can be returned. Arguments to a function are given names and are listed between the parentheses on the function definition line.

For example, consider a function to determine the minimum of two numbers:

```
function minimum(a, b)
   if (a < b) return a else return b
end

' call the function
count = 129
outln("Minimum: " + minimum( count, 77))
```

In SamUL, changing the value of a function's named arguments will modify the variable in the calling program. Instead of passing the actual value of a parameter `a`, SamUL always passes a *reference* to the variable in the original program. The reference is hidden from the user, so the variable acts just like any other variable inside the function.

Because arguments are passed by reference (as in Fortran, for example), a function can "return" more than one value. For example:

```
function sumdiffmult(s, d, a, b)
   s = a+b
   d = a-b
   return a*b
end

sum = -1
diff = -1
mult = sumdiffmult(sum, diff, 20, 7)

outln("Sum: " + sum + " Diff: " + diff + " Mult: " + mult) ' will output 27, 13, and 140
```

5.1.4 Variable Scope

Generally, variables used inside a function are considered "local", and cannot be accessed from the caller program. For example:

```
function triple(x)
   y = 3*x
end

triple( 4 )
```

```
outln( y ) ' this will fail because y is local to the triple function
```

As we have seen, we can write useful functions using arguments and return values to pass data into and out of functions. However, sometimes there are some many inputs to a function that it becomes very cumbersome to list them all as arguments. Alternatively, you might have some variables that are used throughout your program, or are considered reference values or constants. For these situations, you can define variables to be `global` in SamUL, and then they can be used inside functions and in the main program. For example:

```
global pi = 3.1415926

function circumference( r )
   return 2*pi*r
end

function deg2rad( x )
   return pi/180*x
end

outln( "PI: " + pi )
outln( "CIRC: " + circumference( 3 ) )
outln( "D2R: " + deg2rad( 180 ) )
```

Common programming advice is to minimize the number of global variables used in a program. Sometimes the are certainly necessary, but too many can lead to mistakes that are harder to debug and correct, and can reduce the readability and maintainability of your script.

5.2 Built-in SamUL Functions

Throughout this guide, we have made use of built-in functions like `in`, `outln`, and others. These functions are included with SamUL automatically, and called in exactly the same way as user functions. Like user functions, they can return values, and sometimes they modify the arguments sent to them. Refer to the "Standard Library Reference" at the end of this guide for documentation on each function's capabilities, parameters, and return values.

6 Input, Output, and System Access

SamUL provides a variety of standard library functions to work with files, directories, and interact with other programs. So far, we have used the `in`, `out`, and `outln` functions to accept user input and display program output in the runtime console window. Now we will learn about accessing files and other programs.

6.1 Working with Text Files

To write data to a text file, use the `writetextfile` function. `writetextfile` accepts any type of variable, but most frequently you will write text stored in a string variable. For example:

```
data = ""
for (i=0;i<10;i=i+1) data = data + "Text Data Line " + string(i) + "\n"
ok = writetextfile( "C:/test.txt", data )
if (not ok) outln("Error writing text file.")
```

Reading a text file is just as simple with the `readtextfile` function.

```
mytext = ""
if (not readtextfile( "C:/test.txt", mytext ))
   outln("could not read text file.")
else
   outln("text data:")
   out(mytext)
end
```

While these functions offer an easy way to read an entire text file, often it is useful to be able to access it line by line. SamUL provides the `open`, `close`, and `readln` functions for this purpose.

```
file = open("c:/test.txt", "r")
if (not file)
   outln("could not open file")
   exit
end

declare line
while ( readln( file, line ) )
   outln( "My Text Line='" + line + "'" )
end

close(file)
```

In the example above, `file` is a number that represents the file on the disk. The `open` function opens the specified file for reading when the `"r"` parameter is given. The `readln` function will return `true` as long as there are more lines to be read from the file, and the text of each line is placed in the `line` variable.

Another way to access individual lines of a text file uses the `split` function to return an array of text lines. For example:

```
mytext = ""
readtextfile( "C:/test.txt", mytext )
lines = split( mytext, "\n" )
outln("There are " + length(lines) + " lines of text in the file.")
if (length(lines) > 5) outln("Line 5: '", lines[5], "'")
```

6.2 File System Functions

Suppose you want to run SAM with many different weather files, and consequently need a list of all the files in a folder that have the *.tm2* extension. SamUL provides the `directorylist` function to help out in this situation. If you want to filter for multiple file extensions, separate them with commas.

```
file_names = directorylist( "C:/Windows", "dll" ) ' could also use "txt,dll"
outln("Found " + length(file_names) + " files that match.")
outln(unsplit(file_names, "\n"))
```

To list all the files in the given folder, leave the extension string empty or pass in `"*"`.

Sometimes you need to be able to quickly extract the file name from the full path, or vice versa. The functions `filenameonly` and `dirnameonly` extract the respective sections of the file name, returning the result.

To test whether a file or directory exist, use the `direxists` or `fileexists` functions. Examples:

```
path = "C:/SAM/2010.11.9/samsim.dll"
dir = dirnameonly( path )
name = filenameonly( path )
outln( "Path: " + path )
outln( "Name: " + name + " Exists? " + fileexists(path) )
outln( "Dir: " + dir + " Exists? " + direxists(dir))
```

6.3 Standard Dialogs

To facilitate writing more interactive scripts, SamUL includes various dialog functions. We have already used the `notice` and `yesno` functions in previous examples.

The `choosefile` function pops up a file selection dialog to the user, prompting them to select a file. `choosefile` will accept three optional parameters: the path of the initial directory to show in the dialog, a wildcard filter like `"*.txt"` to limit the types of files shown in the list, and a dialog caption to display on the window. Example:

```
file = choosefile("c:/SAM", "*.dll", "Choose a DLL file")
if (file == "")
   notice("You did not choose a file, quitting.")
   exit
else
   if ( not yesno("Do you want to load:\n\n" + file)) exit

   ' proceed to load .dll file
   outln("Loading " + file)
end
```

6.4 Calling Other Programs

Suppose you have a program on your computer that reads an input file, makes some complicated calculations, and writes an output file. For example, a program could read in some system specifications and calculate its heat loss coefficients that could be used in a SAM analysis.

There are two very similar ways to call external programs: the `system` and `shell` functions. They are identical except that `shell` pops up an interactive system command window and runs the program in it. Both functions will wait until the called program finishes before returning to SamUL, so that the program runs *synchronously*. Examples:

```
system("notepad.exe") ' run notepad and wait
shell("ipconfig /all > c:/test.txt") ' run in the system shell
output = ""
readtextfile("c:/test.txt", output)
outln(output)
```

Each program runs in a folder that the program refers to as the *working directory*. Sometimes you may need to switch the working directory to conveniently access other files, or to allow an external program to run correctly.

```
working_dir = cwd()    ' get the current working directory
chdir( "C:/windows" )  ' change the working directory
outln("cwd=" + cwd() )
chdir(working_dir)     ' change it back to the original one
outln("cwd=" + cwd() )
```

7 Interfacing With SAM Analyses

The SamUL language would be of little interest if it did not allow for direct manipulation and automation of SAM analyses. To this end, there is a set of included function calls that can set SAM input variables, invoke a simulation, and retrieve output data.

All of the SamUL function calls involve only "base case" analysis. That is, the built-in parametrics, sensitivity, optimization, and statistical simulation types that are controlled from the user interface are not accessible from SamUL. This is probably less of a hindrance than it sounds, as SamUL exists primarily to allow for specialized simulations that do not fall into one of those categories.

7.1 Getting Started

SamUL scripts are part of a SAM project file that can consist of multiple cases and scripts. The general methodology is to have a SAM case that more or less describes the system you want to investigate, and then create a SamUL script within the same project that can manipulate the case. A SamUL script can only operate on one case at a time, and the *active* case is specified using the `SetActiveCase` function call. For example:

```
SetActiveCase("PV System in Arizona Case")
```

7.2 Changing Input Values

Once an "active" case has been chosen, you can change base case input values using the `SetInput` function. If an input affects other calculated variables, they are automatically recalculated, and the updated values are shown on the case input page. Calling `SetInput` causes the SAM interface to be updated just as if the user had changed the variable manually. Examples:

```
SetInput( "system.degradation", 12.5 )   ' set degradation to 12.5 \%/year
SetInput( "pvwatts.array_type", 1 ) ' sets PVWatts array tracking mode to one axis
```

SamUL requires that you provide the internal names of variables to access them. These names can be accessed from the third button on the SamUL toolbar. A dialog box will pop up, listing all SAM variables sorted by grouping and labels. The internal data type of a variable is also listed. Simply select the input variable from the hierarchical menu, and the internal name will be pasted into the SamUL script at the cursor position.

Unfortunately, because of the huge number of variables in SAM, there is no comprehensive reference manual that describes each variable's values or any special formatting that may be required. For example, the PV shading derate factor matrix is actually stored in memory as a one-dimensional column-major array, with the first two elements indicating the number of rows and columns respectively. Supposing that you had read in a 2D array from a text file into the `shading[i,j]` variable, you must convert it to a single dimensional array representation as below:

```
shadarr = allocate(12*24+2)
shadarr[0] = 12
shadarr[1] = 24
c=2
for (i=0; i<12; i=i+1)
   for (j=0; j<24; j=j+1)
     shadarr[c] = shading[i,j]
     c=c+1
   end
end
```

Then you can write `setinput("pv.shading.mxh.factors?, shadarr)"` and it will assign the factors correctly. The same holds true for most 2D matrix representations in SAM.

One exception is the heliostat field layout matrix for CSP Power Towers. The matrix is also stored as a one-dimensional array as described above, but there is one more number attached to the end of the array to hold the span angle in degrees. Thus the total array length is nrows*ncols+3. In any situation, it is always possible to call the `GetInput` with a variable to inspect how the data is stored.

```
setactivecase("New CSP Power Tower Case 1")

x = getinput("csp.pt.sf.user_field")
outln( "len=",length(x),": ",x)
```

7.3 Simulating and Saving Output

To start a base case simulation, use the `simulate` function. It take a boolean parameter to specify whether to save the hourly (8760) outputs from the simulation. After the simulation has finished, you can access the outputs using the `getoutput` function.

```
setactivecase( "Residential PV System" )
setinput( "system.degradation", 12.5 )   ' set degradation to 12.5 percent
simulate( )
lcoe = getoutput("sv.lcoe_real")
notice( "LCOE = " + lcoe )
```

As with the input variables, the internal variable names of the available outputs are also accessible from the SamUL toolbar.

You can also save several outputs to a comma-separated value (CSV) file to work with in Excel or another program using the `writeresults` function. The outputs variables are passed to the function separated by commas in a single string, and each variable is dumped as a separate column in the CSV file.

```
setactivecase( "Residential PV System" )
setinput( "system.degradation", 12.5 )   ' set degradation to 12.5 percent
simulate( )
writeresults( "c:/test.csv", "system.hourly.e_net,system.monthly.e_net,sv.lcoe_nom")
```

7.4 Batching Weather Files

Let's return to the original hypothetical example discussed in the introduction. You have directory of weather files, and for each one you are asked to calculate the hourly generation and LCOE for the system.

The code addresses this need, and makes use of many SamUL language capabilities and built-in function calls.

```
' Set the active case from the current ones
setactivecase("Simple PV System")

' Specify a directory to use for weather batching
dir = "c:/Documents and Settings/David Smith/Desktop/Weather Files"

' List all the files in a directory that have the extension "tm2"
file_list = DirectoryList(dir, "tm2")

' loop through all the files
count = length(file_list)
for (i=0;i<count;i=i+1)

   out("Weather (",(i+1)," of ",count,")="+FileNameOnly(file_list[i])+"\n")
```

```
   ' set the climate variable to the file name
   setinput("climate.location", file_list[i])

   ' run the base case
   simulate( )

   ' make the output file name
   output_file = dir + "/output_" + FileNameOnly(file_list[i]) + ".csv"

   out("Writing Output File: "+FileNameOnly(output_file)+"\n\n")

   ' dump the needed results into a CSV file
   WriteResults(output_file, "system.hourly.e_net,sv.lcoe_nom")
end
```

This example and several others are included in the standard SAM sample files.

8 Case Study: PV System Database Processing

8.1 Problem Description

Suppose your supervisor sends you an Excel spreadsheet with a list of 30 photovoltaic systems at various locations. For each system, you are given a street address, system size in kW, capital cost, and a flat utility rate in cents/kWh for the location. You are asked to add additional columns to the spreadsheet to show the levelized cost of electricity (LCOE) in both real and nominal terms, the total annual energy production, the payback period if the system were installed. For each system, the array tilt should equal the location's latitude. The spreadsheet may look something like:

Address	PV System Size	Capital Cost	Utility Rate
300 West Second Street Little Rock Ar 72000	170	929675	7.96
15902 Jamaica Ave Jamaica NY	700	3800000	16.40
125 South Grand Avenue Pasadena CA 91105	125	330000	15.75
135 High Street Hartford CT 06103	355	1900000	16.64
333 Constitution Avenue NW Washington DC 20001	970	5200000	13.76
717 Madison Place NW Washington DC 20005	200	1090000	13.76

Table 5: Input Data Table (.csv file format)

One approach would be to calculate all of the metrics by manually setting up each system in SAM. Another possibility would be to set up a complicated linked parametric simulation for all the systems, after having downloaded the correct weather data for each location. Either way, the task would be at best cumbersome and error prone. Worse, if your supervisor decides to change one or two inputs, or later wants additional output metrics, it would be a hassle to start all over again. SamUL provides all the tools you need to automate this process in a flexible way.

8.2 Obtaining Latitudes and Longitudes

The first task is to determine the latitude and longitude of each address, so that we can find appropriate hourly weather data for each location. Google conveniently provides a simple web service for decoding a street address into coordinates. We simply need a valid URL string to pass to Google, and it will return the latitude and longitude in a string via the HTTP/GET method. SamUL has a built-in function `httpget` to process a GET request and returns the string data from the server.

We will code up this functionality in a SamUL function called `addr2latlon` that fills in latitude and longitude from an address. The first task is to normalize the address string to first remove multiple spaces, and then replace single spaces with the `+` sign to make a valid URL. The result of `httpget` is split at each comma location, and the 3rd and 4th elements (latitude and longitude) are extracted and converted to double precision numbers.

If the web service returned something strange, the latitude and longitude are set to -1 to signify an error. The complete function is listed below.

```
function addr2latlon( address, lat, lon )
   addr = address
   strreplace( addr, "   ", " ")
   strreplace( addr, "  ", " ")
   strreplace( addr, " ", "+" )
   query = "http://maps.google.com/maps/geo?q=" + addr + "&output=csv&key=mykey&sensor=false"
   result = httpget( query )
   parts = split( result, "," )
   if (length(parts) == 4)
      lat = double( parts[2] )
      lon = double( parts[3] )
   else
      lat = -1
      lon = -1
   end
end
```

8.3 Downloading Weather Data

Hourly TMY2 formatted weather data can be retrieved from the Mercator web service provided by NREL. Since we are not doing analysis for a particular year, we will write a function to download the typical (TDY) weather data for a location. The `getweather` function uses the `addr2latlon` function listed above to translate the address to coordinates. Given a latitude and longitude, the proper 10 km grid cell can be determined using the `translate` function listed below. The details of this translation will not be covered here, and are specific to how NREL stores the weather files on the server.

Using the grid code and directory information returned from `translate`, we can create a URL for

the weather data file that we wish to download. Using the built-in SamUL function `httpdownload`, we download the compressed weather file to a local folder on our computer. Then, it is decompressed and the .gz extension removed using the `decompress` function. If the decompression succeeds, the compressed file is deleted, and we are left with the proper .tm2 weather data file that can be processed by SAM.

```
function getweather( addr, targetdir, weatherfile, lat, lon )

   lat = -1.0
   lon = -1.0
   grid = ""
   dir = ""

   addr2latlon(addr, lat, lon)
   translate( lat, lon, grid, dir )

   local = targetdir + "/weather_" + grid + ".tm2.gz"

   url = "http://mercator.nrel.gov/perez_tdy/" + dir + "/radwx_" + grid + "_9805.tm2.gz"

   if ( not httpdownload( url, local ) )
      outln("failed to download\n\t" + url + "\n\t" + local )
      return false
   end

   weatherfile = strleft(local, strlen(local)-3)

   if (not decompress( local, weatherfile ))
      outln("failed to decompress file.")
      return false
   end

   deletefile( local )

   return true
end

function translate( lat, lon, gridcode, dir )
   short_lat = integer(abs( lat ))
   short_lon = integer(abs( lon ))

   if ( mod(short_lat,2) > 0 ) short_lat = short_lat-1
   if ( mod(short_lon,2) > 0 ) short_lon = short_lon-1
   short_lon = short_lon + 2

   dir = short_lon + "" + short_lat
```

```
   glat = string( integer( abs(lat)*10 )) + "5"
   glon = string( integer( abs(lon)*10 )) + "5"

   if (strlen(glon)<5) glon = "0" + glon
   gridcode = glon + glat
end
```

8.4 Processing the Input File

Now that we have access to weather data for each of the locations in the spreadsheet, the next task is to read it in and perform each simulation. After the simulation for each location is finished, we will immediately write the results to another file. SamUL conveniently allows multiple files to be open at the same time.

The code below opens the input and output files, and scans each line of the input for the various input parameters, runs a simulation, and writes the results to an output file.

```
setactivecase("Baseline PV System")

workdir = "/Users/adobos/Desktop"

input = open( workdir + "/input.csv", "r" )
if (not input)
   outln("could not open input file")
   exit
end

output = open( workdir + "/output.csv", "w" )
if (not output)
   outln("could not open output file")
   exit
end

write( output, "lat,lon,lcoe nominal,lcoe real,enet,payback\n" )

declare buffer
readln(input, buffer)
while( readln( input, buffer ) )
   cols = split( buffer, "," )
   address = cols[0]
   size = cols[1]
   cost = cols[2]
   rate = cols[3]

   declare lat = -1.0,lon = -1.0, weatherfile=""
```

```
   if (not getweather( address, workdir, weatherfile, lat, lon ))
      outln("failed to get weather data for: " + address)
      exit
   end

   module_cost = cost / ( size * 1000 )

   setinput( "climate.location", weatherfile )
   setinput( "pvwatts.dcrate", size )
   setinput( "pvwatts.tilt", abs(lat) )
   setinput( "pv.cost.per_module", module_cost )
   setinput( "ur.flat.buy_rate", rate/100.0 )

   outln("Simulating location: " + address)
   simulate()

   declare lcoe_nom=0,lcoe_real=0,enet=0,payback=0
   lcoe_nom = getoutput("sv.lcoe_nom")
   lcoe_real = getoutput("sv.lcoe_real")
   enet = getoutput( "sv.annual_output" )
   payback = getoutput( "sv.payback" )

   write(output, lat+","+lon+","+lcoe_nom+","+lcoe_real+","+enet+","+payback)
end

close(input)
close(output)
```

In this example, the cost data for each location represented the total installed system cost. However, SAM does not have a direct input for the total cost, as it is calculated from many different cost components. To trick SAM into using the total cost that we have, however, we simply set all the costs to zero, and calculate an effective DC module cost given the nameplate size of the system and the total cost. Then, setting the module cost variable to this value will naturally result in the desired total system cost as well.

This examples shows the power SamUL to automate an otherwise tedious and error prone task. While the data set presented here is relatively small, one could imagine such a task for thousands of locations, system sizes, and costs. The ability to exercise SAM's capabilities via scripting is a tool of core importance in an advanced SAM user's toolbox.

Additional references:

The case study example and several others are included as part of the standard SAM installation. Please browse the examples for additional information, or contact Solar Advisor Support at

`solar.advisor.support@nrel.gov`.

9 Library Reference

9.1 Type/Data Manipulation

`TypeOf`
(<VARIANT>):STRING
Returns a description of the argument type.

`Integer`
(VARIANT):INTEGER
Converts the variable to an integer number.

`Double`
(VARIANT):DOUBLE
Converts the variable to a double-precision floating point number.

`Boolean`
(VARIANT):BOOLEAN
Converts the variable to a boolean.

`String`
(...):STRING
Converts the given variables to a string.

`IntegerArray`
(STRING):ARRAY
Converts a string delimited by {;, \t\n} to an integer array.

`DoubleArray`
(STRING):ARRAY
Converts a string delimited by {;, \t\n} to a double-precision floating point array.

`Length`
(ARRAY):INTEGER
Return the length of an array.

`Array`
(...):ARRAY
Creates an array out of the argument list.

`Allocate`
(INTEGER:PRIMARY, [INTEGER:SECONDARY]):ARRAY
Creates an empty array with the specified dimensions.

`Resize`
(<ARRAY>, INTEGER:PRIMARY, [INTEGER:SECONDARY]):NONE
Resizes an array or 2D matrix.

`Append`
(<ARRAY>, ...):NONE
Appends one or more items to an array.

`Prepend`
(<ARRAY>, ...):NONE
Prepends one or more items to an array.

9.2 Input/Output

`Out`
(...):NONE
Print data to the output device.

`OutLn`
(...):NONE
Print data to the output device followed by a newline.

`Print`
(STRING:Format, ...):NONE
Print formatted data to the output device using an extended 'printf' syntax.

`In`
(...):STRING
Request input from the input device, showing an optional prompt.

`Notice`
(...):NONE
Show a message dialog.

`YesNo`
(...):BOOLEAN)
Show a Yes/No dialog. Returns true if yes was clicked

`ChooseFile`
([STRING:Initial dir], [STRING:Filter], [STRING:Caption]):STRING
Show a file selection dialog, with optional parameters.

`StartTimer`
(NONE):NONE
Starts a stop watch timer.

`ElapsedTime`
(NONE):INTEGER
Returns elapsed milliseconds since last call to 'StartTimer'

`MilliSleep`
(INTEGER:Milliseconds):NONE
Sleep for the specified amount of time.

`DateTime`
(NONE):STRING
Returns the current date and time.

`Open`
(STRING:File, STRING:Mode):INTEGER
Opens a file for reading 'r', writing 'w', or appending 'a'.

`Close`
(INTEGER:FileNum):NONE
Closes a file.

`Seek`
(INTEGER:FileNum, INTEGER:Offset, INTEGER:Origin):INTEGER
Sets the position in an open file.

`Tell`
(INTEGER:FileNum):INTEGER
Returns the current file position.

`Eof`
(INTEGER:FileNum):BOOLEAN
Determines whether a file is at the end.

`Flush`
(INTEGER:FileNum):INTEGER
Flushes the current file object to disk.

`Write`
(INTEGER:FileNum, ...):BOOLEAN
Writes data as text to a file.

`WriteN`
(INTEGER:FileNum, VARIANT data, INTEGER: NumChars):BOOLEAN
Writes character data to a file.

`WriteLn`
(INTEGER:FileNum, VARIANT data):BOOLEAN
Writes a line to a file as a string.

`ReadN`
(INTEGER:FileNum, <STRING>:Data, INTEGER:NumChars):BOOLEAN
Reads characters from a file.

`ReadLn`
(INTEGER:FileNum, <STRING>:Line):BOOLEAN
Reads a line from a file, returning false if no more lines exist.

`ReadFmt`
(INTEGER:filenum, STRING:format=[idgefxsb]*, STRING:delimiters, ... VALUE ARGUMENT LIST):BOOLEAN
Reads a data line from a file with the given sequence of types and delimiters. Number of value arguments must equal number of characters in format string

`OpenWF`
(STRING:file, [ARRAY:header info]):INTEGER
Opens a weather (TM2, TM3, EPW) file for reading.

`ReadWF`

(INTEGER:filenum, ARRAY:y—m—d—h—gh—dn—df—wind—tdry—twet—relhum—pres *or* [INTEGER:y, INTEGER:m, INTEGER:d, INTEGER:h, DOUBLE:gh, DOUBLE:dn, DOUBLE:df, DOUBLE:wind, DOUBLE:tdry, DOUBLE:twet, DOUBLE:relhum, DOUBLE:pres]):BOOLEAN

Reads a line of data from a weather file.

`CustomizeTMY3`

(STRING:Source tmy3 file, STRING:Target tmy3 file, [STRING:Column name=gh—dn—df—tdry—twet—wind—pressure—relhum, ARRAY:Values(8760)]*):BOOLEAN

Overwrites columns of 8760 data in a TMY3 file and writes a new file.

`WFStatistics`

(STRING:File, <DOUBLE>:DN, <DOUBLE>:GH, <DOUBLE>:AMBT, <DOUBLE>:WSPD):BOOLEAN

Extracts annual averages of DN, GH, AmbT, and WSpd

`WriteTextFile`

(STRING:Filename, VARIANT data):BOOLEAN

Writes a file of text data to disk. Returns true on success.

`ReadTextFile`

(STRING:Filename, <STRING>:Data):BOOLEAN

Reads a text file from disk, returning true on success.

`GetHomeDir`

(NONE):STRING

Returns the current user's home directory.

`Cwd`

(NONE):STRING

Returns the current working directory.

`ChDir`

(STRING: Path):BOOLEAN

Change the current working directory.

`DirectoryList`

(STRING:Path, STRING:Comma-separated extensions, [BOOLEAN:Include folders]):ARRAY

Enumerates all the files in a directory that match a comma separated string of extensions.

`System`

(STRING):INTEGER

Run a system command, returning the process exit code.

`Shell`

(STRING):BOOLEAN

Run a system command in a new console window. Returns true on success.

`FileNameOnly`

(STRING:Path):STRING

Returns only the file name portion of a full path.

`DirNameOnly`
(STRING:Path):STRING
Returns only the directory portion of a full path.

`Extension`
(STRING:File):STRING
Returns the extension of a file.

`DirExists`
(STRING:Path):BOOLEAN
Returns true if the specified directory exists.

`FileExists`
(STRING:Path):BOOLEAN
Returns true if the specified file exists.

`CopyFile`
(STRING:File1, STRING:File2):BOOLEAN
Copies file 1 to file 2.

`RenameFile`
(STRING:File1, STRING:File2):BOOLEAN
Renames file 1 to file 2.

`DeleteFile`
(STRING:File):BOOLEAN
Deletes the specified file.

`MkDir`
(STRING:Path):BOOLEAN
Creates a directory including the full path to it.

`RmDir`
(STRING:Path):BOOLEAN
Deletes a directory and everything it contains.

`Decompress`
(STRING:Archive, STRING:Target):BOOLEAN
Decompresses an archive file (ZIP, TAR, TAR.GZ, GZ).

`HttpGet`
(STRING:Url):STRING
Performs an HTTP web query and returns the result as plain text.

`HttpDownload`
(STRING:Url, STRING:LocalFile):BOOLEAN
Downloads a file form the web, showing a progress dialog.

9.3 String Manipulation

`StrPos`
(STRING, STRING:Search):INTEGER
Returns the first position of the search string, or -1 if not found.

`StrRPos`
(STRING, STRING:Search):INTEGER
Returns the first position of the search string from the right, or -1 if not found.

`StrLeft`
(STRING, INTEGER:N):STRING
Returns the left 'N' character string.

`StrRight`
(STRING, INTEGER:N):STRING
Returns the right 'N' character string.

`StrLower`
(STRING):STRING
Returns a lower case version of the string.

`StrUpper`
(STRING):STRING
Returns an upper case version of the string.

`StrMid`
(STRING, INTEGER:Start, [INTEGER:Count]):STRING
Returns the substring from the specified start position, of length 'count'. If 'count' is not supplied, the remainder of the string is returned.

`StrLen`
(STRING):INTEGER
Returns the length of a string.

`StrReplace`
(STRING, STRING:s0, STRING:s1):STRING
Returns a string with all instances of 's0' replaced with 's1'.

`StrCmp`
(STRING:s0, STRING:s1):INTEGER
Case-sensitive comparison. Returns 0 if equal, positive if s0 comes before s1, and negative if s1 comes before s0.

`StrICmp`
(STRING:s0, STRING:s1):INTEGER
Case-insensitive comparison. Returns 0 if equal, positive if s0 comes before s1, and negative if s1 comes before s0.

`StrGCh`
(STRING, INTEGER:position):STRING
Gets the character at the specified position.

`StrSCh`
(STRING, INTEGER:position, STRING:char):NONE
Sets the character at the specified position.

`Split`
(STRING, STRING:delimiters):ARRAY
Splits the string into an array.

`Unsplit`
(ARRAY, STRING:delimiters):STRING
Unsplits an array into a string.

`Format`
(STRING:Format, ...):STRING
Formats data into a string using an extended 'printf' syntax.

9.4 Math

`Mod`
(INTEGER, INTEGER):INTEGER
Returns the remainder after X is divided by Y

`Abs`
(NUMBER):NUMBER
Absolute value of the number.

`Min`
(NUMBER, NUMBER *or* ARRAY):NUMBER
Returns the smaller of two values, or the smallest in an array.

`Max`
(NUMBER, NUMBER *or* ARRAY):NUMBER
Returns the larger of two values, or the largest in an array

`Ceil`
(NUMBER):DOUBLE
Returns the number rounded up to the nearest integer.

`Floor`
(NUMBER):DOUBLE
Returns the number rounded down to the nearest integer.

`Sqrt`
(NUMBER):DOUBLE
Returns the square root of a number.

`Pow`
(NUMBER:X, NUMBER:Y):DOUBLE
Returns 'X' raised to the 'Y' power.

`Exp`
(NUMBER):DOUBLE
Returns the exponential value, base 'e'.

`Log`
(NUMBER):DOUBLE
Returns the logarithm of a number, base 'e'.

`Log10`
(NUMBER):DOUBLE
Returns the logarithm of a number, base 10.

`Sin`
(NUMBER):DOUBLE
Returns the sine of a radian value.

`Cos`
(NUMBER):DOUBLE
Returns the cosine of a radian value.

`Tan`
(NUMBER):DOUBLE
Returns the tangent of a radian value.

`ASin`
(NUMBER):DOUBLE
Returns the arcsine of a number in radians.

`ACos`
(NUMBER):DOUBLE
Returns the arccosine of a number in radians.

`ATan`
(NUMBER):DOUBLE
Returns the arctangent of a number in radians.

`ATan2`
(NUMBER:Y, NUMBER:X):DOUBLE
Returns the arctangent of 'Y'/'X' in radians.

`IsNan`
(DOUBLE):BOOLEAN
Returns true if the number is NAN.

`NanVal`
(NONE):DOUBLE
Returns NAN.

`UnifRand`
(NONE):DOUBLE
Returns a random number with uniform distribution (0..1).

`NormRand`
(NONE):DOUBLE
Returns a random number with normal distribution around 0.

9.5 SAM Functions

`SetInput`
(STRING:Variable name, VARIANT value):NONE
Sets an input in the active case.

`GetInput`
(STRING:Variable name):VARIANT
Returns an input value from the active case.

`GetOutput`
(STRING:Variable name):ARRAY
Returns a base case output from the active case's results as a double-precision array.

`ResetOutputSource`
(NONE or STRING:Simulation name, INTEGER:Run number):NONE
Resets the output data source to default BASE case, or changes it to a different simulation and run number.

`ClearSimResults`
(STRING:Simulation name):NONE
Clears all results for the specific simulation name.

`SetActiveCase`
(STRING:Case name):NONE
Sets the active case.

`GetActiveCase`
(NONE):STRING
Returns the active case name.

`SwitchToCase`
(NONE):NONE
Switches to the active case tab in the interface.

`ChangeConfig`
(STRING:Technology, STRING:Financing):BOOLEAN
Changes the current case's configuration. Application must be '*'.

`ListCases`
(NONE):ARRAY
Lists all the cases in the project.

`ProjectFile`
(NONE):STRING
Returns the current project file name.

`AppYield`
(NONE):NONE
Yields the interface to respond to user input.

`SaveProject`
(NONE): BOOLEAN
Saves the project.

`SaveProjectAs`
(STRING):BOOLEAN
Saves the project to the specified file.

`Simulate`
(BOOLEAN:Save hourly data):NONE
Runs a base case simulation with the current inputs, with the option of saving hourly results.

`MPSimulate`
(STRING:Simulation name, ARRAY[ARRAY]:Variable name/value table NRUNS+1 x NVARS with top row having var names):BOOLEAN
Runs many simulations using multiple processors.

`WriteResults`
(STRING:File name, STRING:Comma-separated output variable names):BOOLEAN
Write a comma-separated-value file, with each column specified by a string of comma-separated output names.

`ClearResults`
(NONE):NONE
Clear the active case's results from memory.

`ClearCache`
(NONE):NONE
Clear the memory cache of previously run simulations.

`DeleteTempFiles`
(NONE):NONE
Delete any lingering simulation temporary files.

`SetTimestep`
(STRING:Timestep with units):NONE
Sets the TRNSYS timestep for the active case.

`ReloadDefaults`
(NONE):NONE
Reloads all default values for the active case.

`ListTechnologies`
(NONE):ARRAY
Returns an array of all the technologies in SAM.

`ListFinancing`
(STRING:Technology):ARRAY

Lists all financing options in SAM for a given technology.

`TechnologyType`
(NONE):STRING
Returns the active case technology type.

`FinancingType`
(NONE):STRING
Returns the active case financing type.

`ActiveVariables`
([STRING:Technology, STRING:Financing]):ARRAY
List all active variables for the current case or technology/market name.

`FluidDensity`
(INTEGER:Fluid number, DOUBLE:Temp 'C):DOUBLE
Returns density at temperature Tc for a given fluid number (pressure assumed 1Pa).

`FluidSpecificHeat`
(INTEGER:Fluid number, DOUBLE:Temp 'C):DOUBLE
Returns specific heat at temperature Tc for a given fluid number (pressure assumed 1Pa).

`FluidName`
(INTEGER:Fluid number):STRING
Returns fluid name for a given fluid number.

`PtOptimize`
(NONE):NONE
Optimizes the power tower heliostat field, tower height, receiver height, and receiver diameter.

`LHSCreate`
(NONE):INTEGER
Creates a new Latin Hypercube Sampling object.

`LHSFree`
(INTEGER:lhsref):NONE
Frees an LHS object.

`LHSReset`
(INTEGER:lhsref):NONE
Erases all distributions and correlations in an LHS object.

`LHSSeed`
(INTEGER:seed):NONE
Sets the seed value for the LHS object.

`LHSPoints`
(INTEGER:lhsref, INTEGER:number of points):NONE
Sets the number of samples desired.

`LHSDist`
(INTEGER:lhsref, STRING:distribution name, STRING: variable name, [DOUBLE:param1, DOUBLE:param2, DOU-

BLE:param3, DOUBLE:param4]): NONE
Sets up a distribution for a variable.

`LHSCorr`
(INTEGER:lhsref, STRING:variable 1, STRING:variable 2, DOUBLE:corr val):NONE
Sets up correlation between two variables.

`LHSRun`
(INTEGER:lhsref):BOOLEAN
Runs the LHS sampling program.

`LHSError`
(INTEGER:lhsref):STRING
Returns an error message if any.

`LHSVector`
(INTEGER:lhsref, STRING:variable):ARRAY
Returns the sampled values for a variable.

`STEPCreate`
(NONE):INTEGER
Create a new STEPWISE regression analysis object.

`STEPFree`
(INTEGER:stpref):NONE
Frees a STEPWISE object.

`STEPInput`
(INTEGER:stpref, STRING:name, ARRAY:values):NONE
Sets a STEPWISE input vector.

`STEPOutput`
(INTEGER:stpref, ARRAY:values):NONE
Sets a STEPWISE output vector.

`STEPRun`
(INTEGER:stpref):NONE
Runs the STEPWISE analysis.

`STEPError`
(INTEGER:stpref):STRING
Returns any error code from STEPWISE.

`STEPResult`
(INTEGER:stpref, STRING:name):ARRAY
Returns R2 and SRC for a given input name.

`OpenEIListUtilities`
(NONE):ARRAY
Returns a list of utility company names from OpenEI.org

`OpenEIListRates`
(STRING:Utility name, <ARRAY:Names>, <ARRAY:Guids>):INTEGER

Lists all rate schedules for a utility company.

`OpenEIApplyRate`
(STRING:Guid):BOOLEAN
Downloads and applies the specified rate schedule from OpenEI.

`URdbFileWrite`
(STRING:file):BOOLEAN
Writes a local URdb format file with the current case's utility rate information.

`URdbFileRead`
(STRING:file):BOOLEAN
Reads a local URdb format file and overwrites the current case's utility rate information.